Relazione per "Papas Burgeria"

Casamenti Michele, Kravchuk Paulo, Ranzari Gabriele

30 giugno 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il nostro gruppo ha come obiettivo quello di sviluppare un'implementazione in Java del celebre flash game Papa's Burgeria, mantenendo l'idea originale nel miglior modo possibile. Il gioco simula la gestione di un ristorante di hamburger, mettendo il giocatore nei panni di un cuoco e gestore che si occupa di tutti gli step del servizio offerto: dall'accoglienza iniziale dei clienti alla presa degli ordini stessi, fino alla preparazione degli hamburger e della loro vendita.

Con il procedimento del gioco, ovvero man mano che le giornate vanno avanti all'interno di esso, la difficoltà aumenta con un numero sempre più elevato di clienti e ordini da gestire contemporaneamente. Inoltre, man mano che si va avanti il giocatore riesce a sbloccare nuovi ingredienti andando a complicare di fatto gli ordini futuri.

Il giocatore guadagnerà soldi per ogni panino in base al rating che esso riceve, il rating è dato dalla precisione con cui il giocatore svolge il proprio lavoro ed è importante solamente se il denaro gli è di interesse. Attraverso i soldi, il giocatore può comprare potenziamenti che andranno ad alleviare la difficoltà del gioco.

Requisiti funzionali

- Il giocatore interagirà con i clienti prendendo le istruzioni sul come l'hamburger verrà cotto e con quali ingredienti verrà condito;
- L'ordine verrà visualizzato su un foglio di carta, che farà da guida al giocatore;

- La cottura della carne verrà gestita con una piastra a tempo. Il patty di carne verrà posizionato su di essa e più il tempo scorrerà, più la carne sarà cotta. Sarà necessario cuocere entrambi i lati del patty separatamente;
- La stazione di assemblaggio dell'hamburger dovrà offrire la possibilità al giocatore di ricreare la ricetta, trasportando il patty dalla griglia alla stazione;
- L'assemblaggio inizierà con il pezzo inferiore del pane e terminerà con quello superiore. Inserito un ingrediente, sarà anche possibile rimuoverlo;
- La ricompensa (in denaro) del cliente deve essere calcolata tenendo in considerazione l'accuratezza con la quale viene realizzato il panino;
- La difficoltà dovrà incrementare con lo scorrere delle giornate. Verranno sbloccati più ingredienti e compariranno più clienti contemporaneamente;
- A fine giornata, verrà data la possibilità di utilizzare il negozio. Si potranno comprare potenziamenti utili a semplificare il gioco;

Requisiti non funzionali

- Il gioco dovrà essere funzionare su più sistemi operativi, sia nell'esecuzione sia nella gestione di risorse e dei file relativi ai salvataggi.
- L'esperienza di gioco deve risultare fluida e intuitiva.
- L'interfaccia grafica deve essere adattabile a schermi di diverse misure e al ridimensionamento della finestra da parte dell'utente.

1.2 Modello del Dominio

Il gioco è un cooking game dove il giocatore va a gestire un ristorante avendo a disposizione una moltitudine di funzioni. In linea generale, il giocatore dovrà essere in grado di cambiare schermata, prendere gli ordini, cuocere i patty, assemblare il panino e consegnarlo. Di seguito vi sono elencati gli oggetti principali che rappresentano in modo astratto il dominio di funzionalità previste all'interno del gioco:

- `Scene`: rappresenta una singola scena all'interno del modello, ovvero un'entità astratta che definisce un contesto in cui avvengono specifiche interazioni tra le altre entità;
  - `Register`: rappresenta la scena in cui il cliente effettua un'ordinazione;
  - `Grill`: rappresenta la scena dedicata alla cottura della carne;
  - `BurgerAssembly`: rappresenta la scena in cui gli ingredienti vengono assemblati per formare l'hamburger completo;
- `Hamburger`: rappresenta l'entità del panino, composta da un insieme di ingredienti. Ogni hamburger contiene almeno due istanze di `Ingredient`, corrispondenti alle due parti principali del panino: la base e il coperchio del pane;
- `Ingredient`: rappresenta un singolo ingrediente che può essere utilizzato per comporre un hamburger.
- `Patty`: rappresenta un'entità specializzata di tipo `Ingredient`, ovvero la carne. Estende le funzionalità base di `Ingredient` includendo informazioni aggiuntive sulla cottura.
- `Order`: rappresenta l'entità relativa all'ordinazione effettuata da un cliente. Contiene le informazioni sul panino ordinato, facendo riferimento a un oggetto di tipo `Hamburger`. Ogni ordine è associato a un solo hamburger e a un solo customer.
- `Customer`: rappresenta l'entità del cliente che effettua un'ordinazione. Ogni cliente mantiene una relazione con un oggetto di tipo `Order`, che identifica l'hamburger ordinato.

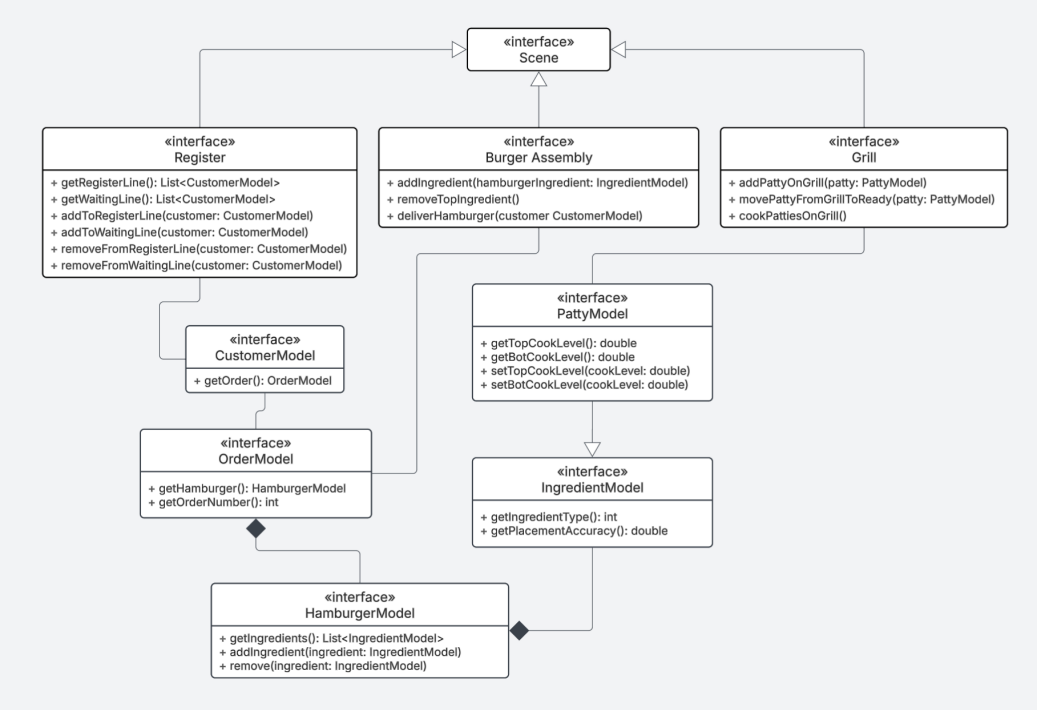


Figura 1.1: : Schema UML generale che rappresenta le principali interfacce nel gioco.

Capitolo 2

Design

2.1 Architettura

Il software è stato strutturato utilizzando il pattern architetturale *Model-View-Controller* (MVC), con l'obiettivo di ottenere una chiara separazione delle responsabilità tra i diversi livelli dell'architettura, facilitare l'estensione e la manutenzione del programma, e rendere possibile la sostituzione delle implementazioni di singoli componenti (ad esempio quelli del layer *View*) senza impatti o modifiche sulle altre parti del sistema.

- Struttura generale (layer principali):
  - `Model`: questo layer rappresenta il dominio dell'applicazione. Contiene sia le entità principali descritte nella fase di modellazione, sia eventuali specializzazioni e aggiunte emerse durante la progettazione architetturale e l'implementazione. Le classi appartenenti a questo layer si occupano esclusivamente di rappresentare e gestire lo stato del gioco, e non presentano alcun riferimento né al *Controller* né alla *View*.
  - `Controller`: funge da mediatore tra la *View* e il *Model*. Riceve gli input dalla *View* e interagisce con il *Model* per aggiornare lo stato dell'applicazione o recuperarne i dati, comunicando successivamente alla *View* quali informazioni visualizzare. Ogni componente all'interno del layer `Controller` conosce i modelli di dominio di cui necessita, ma resta completamente indipendente dalle implementazioni del layer *View*.
  - `View`: questo layer gestisce l'interfaccia utente, mostrando le informazioni ottenute tramite i *Controller* e interagendo con essi per

gestire l'input dell'utente, delegando le operazioni da eseguire. Le implementazioni presenti in questo layer non contengono logica applicativa e non accedono direttamente al *Model*, in modo da garantire che la sostituzione della *View* non comporti modifiche nei layer sottostanti.

- Componente aggiuntivo (layer intermedio):
  - `Service`: layer intermedio in cui vengono incapsulate logiche comuni o trasversali non attribuibili in modo diretto né al *Controller* né al *Model*. Tali logiche sono progettate con un livello di astrazione sufficiente a non violare i confini tra i diversi layer, consentendo l'utilizzo delle implementazioni di *Service* da parte di più componenti dell'architettura in modo controllato e coerente.

L'architettura, a livello concettuale, presenta una scomposizione modulare in *scene indipendenti*. Ogni astrazione di *scena* rappresenta un contesto funzionale isolato dell'applicazione (es. cassa, griglia, assemblaggio) e incapsula la propria terna di componenti: *Model*, *Controller* e *View*.

Questa organizzazione consente di separare le logiche, evitando di appesantire i controller con codice non pertinente, mantenendoli semplici e riducendo collegamenti incrociati al minimo indispensabile.

Il componente `GameView` funge da entry point all'interno di questa struttura: si occupa di selezionare dinamicamente la *View* appropriata da mostrare all'utente, attraverso un callback delegato al componente `SceneService`. La scelta della scena da attivare è responsabilità del *Controller*, che interagisce con il `SceneService` tramite l'astrazione del concetto di scena, mantenendo così il disaccoppiamento dal layer di *View*.

L'intero sistema è coordinato tramite `Google Guice`, sfruttato per semplificare la *dependency injection* e rafforzare l'indipendenza tra layer, contratti e relative le implementazioni. Ad esempio, anche se un *Controller* richiede il cambio scena a `SceneType.GRILL`, è possibile sostituire facilmente l'implementazione della *View* associata a tale tipo senza modificare la logica del controller.

Figura 2.1: : Rappresentazione generalizzata complessiva dell'architettura.

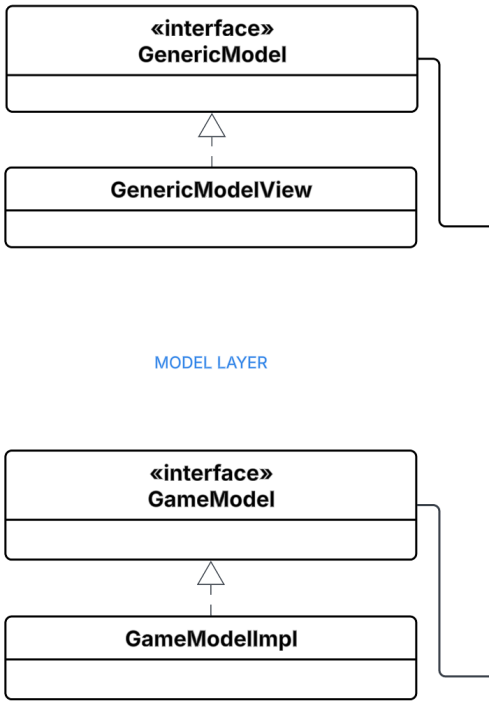


Figura 2.2: : Focus sul Model layer.

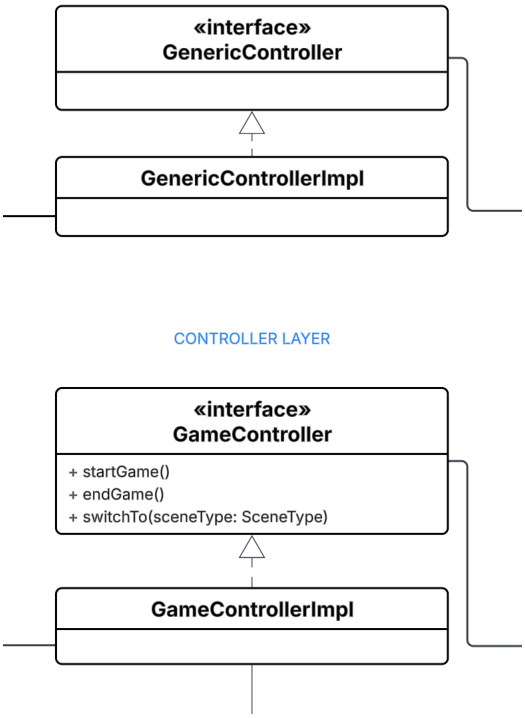


Figura 2.3: : Focus sul Controller layer.

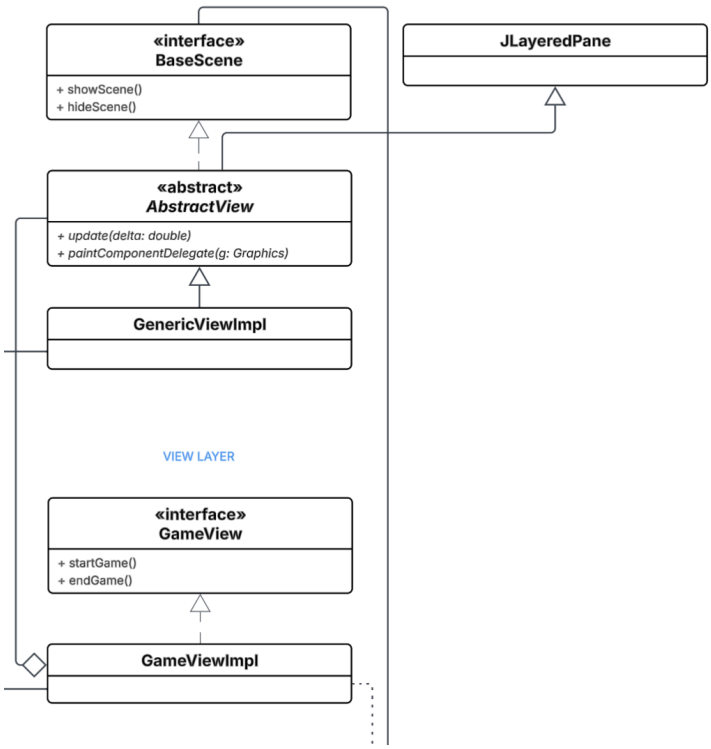


Figura 2.4: : Focus sul View layer.

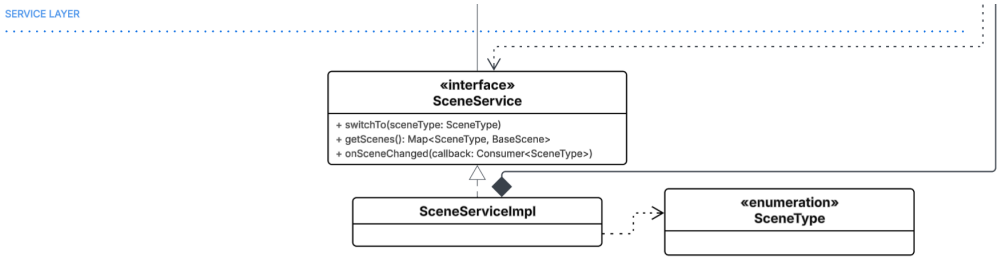


Figura 2.5: : Focus sul Service layer.

2.2 Design dettagliato

2.2.1 Casamenti Michele

Modellazione e stampa delle file di clienti

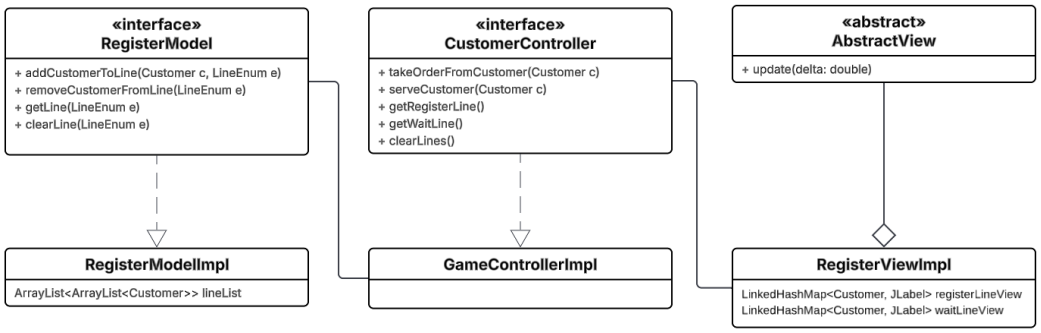


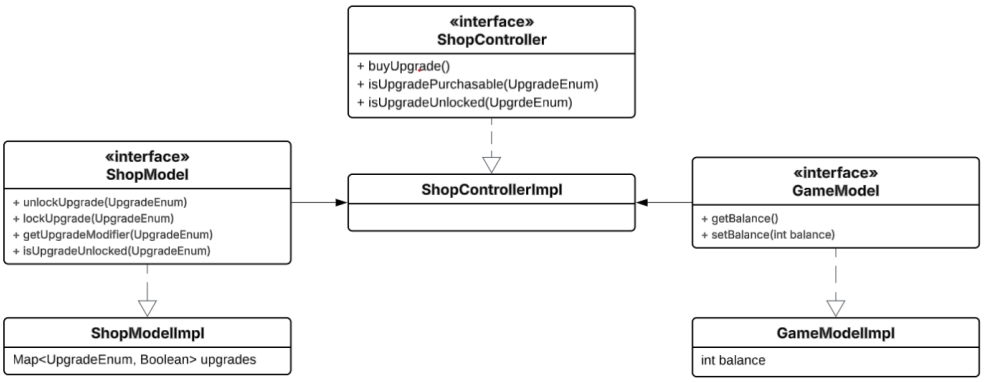
Figura 2.6: Enter Caption

Problema Nel gioco saranno presenti due file di clienti: una dove i clienti attendono che i loro ordini vengano presi e una dove attendono che l'ordine venga a loro consegnato. Le file terranno conto dell'ordine di arrivo dei clienti e il cliente si sposterà dalla prima alla seconda quando il loro ordine viene preso. Sarà necessario quindi creare un modello con due file di clienti e una view che resti aggiornata alla modifica dello stato di queste due file.

Soluzione Viene creato un `RegisterModel` che contiene le due file di clienti (memorizzate come `ArrayList` di `Customer`).Viene creato un `CustomerController`, che modifichi opportunamente le due file presenti nel `RegisterModel` quando viene preso o consegnato un ordine. Viene creata una `RegisterView`, che viene aggiornata periodicamente. La `RegisterView` ha due `LinkedHashMap` <`Customer`, `JLabel`> . Ogni label è identificato dal `Customer` che dovrà stampare. Ad ogni update() della `RegisterView`, viene controllato (per entrambe le file) se i customer presenti nel model (attraverso chiamate del controller) si equivalgono. Qualora i valori dei clienti non coincidano, la lista corrente della `RegisterView` viene svuotata e ri-riempita con i valori presi da controller e da nuovi `JLabel`. Poi i `JLabel` saranno ri-aggiunti al pannello della view.

Pattern utilizzati: MVC

Modellazione del negozio e gestione economia

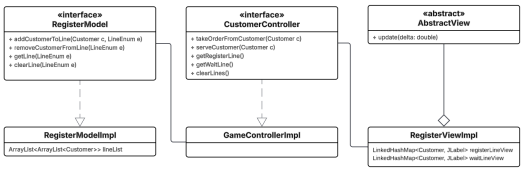


Problema Il gioco ha un negozio e un economia. Nel negozio si possono comprare vari upgrade, che rallentino l'arrivo di clienti, che aumentino i soldi ricavati o che aumentino la tolleranza se l'hamburger viene cucinato male.

Soluzione Gli upgrade vengono salvati come `UpgradeEnum`, e sono composti da: nome, descrizione, costo e modificatore (in valore da 0.0 a 1.0). Lo stato di sbloccamento dell'upgrade sarà salvato in una `Map<UpgradeEnum, boolean>` nella classe `ShopModelImpl`. Mentre il bilancio sarà salvato dentro `GameModelImpl`. Tutte le transizioni vengono effettuate all'interno di uno `ShopControllerImpl`. La classe `ShopModel` potrà inoltre essere richiamata in altri controller per ricavare i modifer e calcolare tutti gli upgrade necessari. La funzione `getUpgradeModifier(UpgradeEnum)` ritorna 0 se l'upgrade è bloccato.

Pattern utilizzati: Enum, Singleton e MVC.

Gestione arrivo dei clienti



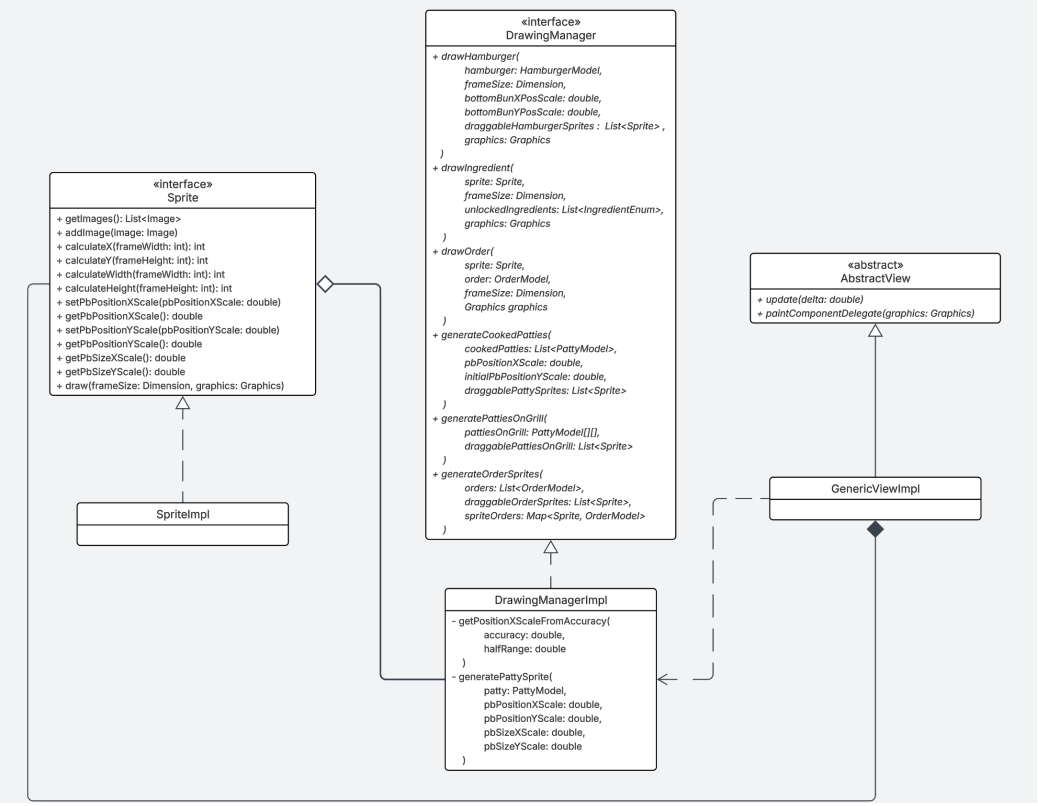
Problema Nel gioco, i clienti devono arrivare mentre il giocatore è impegnato nella cucina a cucinare gli hamburger. Inoltre i clienti devono arrivare con velocità variabile e in numero crescente a seconda degli upgrade sbloccati nel negozio.

Soluzione Il `RegisterModel` ha al suo interno un oggetto `CustomerThread` che implementa Runnable. Il thread viene gestito attraverso interrupt. Il `CustomerThread` periodicamente genera un numero di clienti (il numero di clienti e la frequenza sono definiti dal `RegisterModel`) e variano a seconda della difficoltà della giornata e dagli upgrade comprati nello shop. Il `Thread` chiama la funzione addCustomer del `RegisterModel` e tiene traccia di quanti clienti sono stati generati.

Pattern utilizzati: Thread

2.2.2 Ranzari Gabriele

Disegnare sprite composti da immagini singole o multiple e che si devono sempre adattare allo schermo

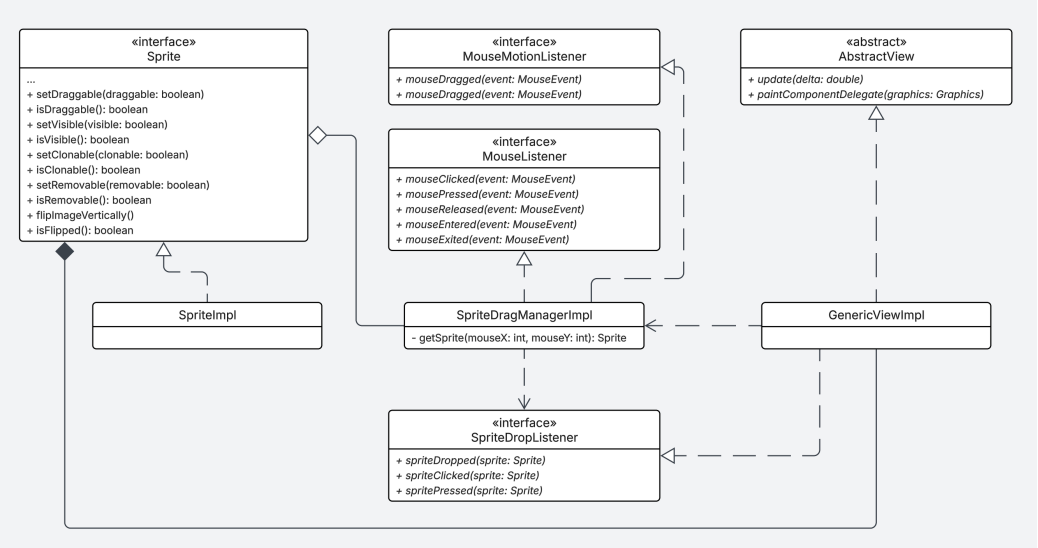


Problema Nel gioco bisogna gestire immagini di due tipi: quelle semplici, come singole fettine di formaggio o di lattuga, e quelle composte, come i patty che presentano più layer sovrapposti per mostrare la cottura e gli effetti di cottura sulla griglia. Ogni sprite deve poi adattarsi dinamicamente alla dimensione del `JComponent` in cui viene disegnato, mantenendo proporzioni e posizionamento relativi allo schermo. Bisogna evitare la duplicazione della logica di caricamento, calcolo dei pixel e disegno per le diverse tipologie, pur garantendo rapidità nel rendering e facilità di estensione quando vengono aggiunti nuovi elementi grafici.

Soluzione Viene creata una classe `Sprite` con una sua interfaccia e implementazione che possiede una lista di immagini e quattro variabili `double` che vanno da 0 a 1, due per le coordinate dello sprite e due per le dimensioni di essa, con dei suoi metodi per settarle e prenderle e altri per calcolare la

posizione e la dimensione in pixel in base alla grandezza del `JComponent` in cui sono inserite. Inoltre per poterle disegnarle visto che si vuole evitare di duplicare codice si crea una classe `DrawingManager` che gestisce la creazione e il disegno di sprite di diversi elementi come ingredienti, hamburger e ordini.

Gestire il drag and drop di Sprite con caratteristiche diverse



Problema In varie view bisogna trascinare i sprite per eseguire varie funzionalità, però non tutti gli sprite sono gestiti allo stesso modo, alcuni non sono trascinabili quindi si deve creare una loro copia e nascondere l'originale, altri invece devono creare una loro copia e fare da generatori di ingredienti, altri ancora non posso essere clonabili, eccetera. In più bisogna gestirli in modo diverso in base a dove vengono rilasciati e in base alla view in cui si è, quindi serve un modo centralizzato per gestire l'interazione tra mouse e sprite che eviti duplicazione di codice nelle varie view ma che permetta allo stesso tempo ad ogni view di gestirle gli sprite in modi diversi.

Soluzione Aggiungere a `Sprite` vari `boolean` per capire in quale caso siamo ovvero:

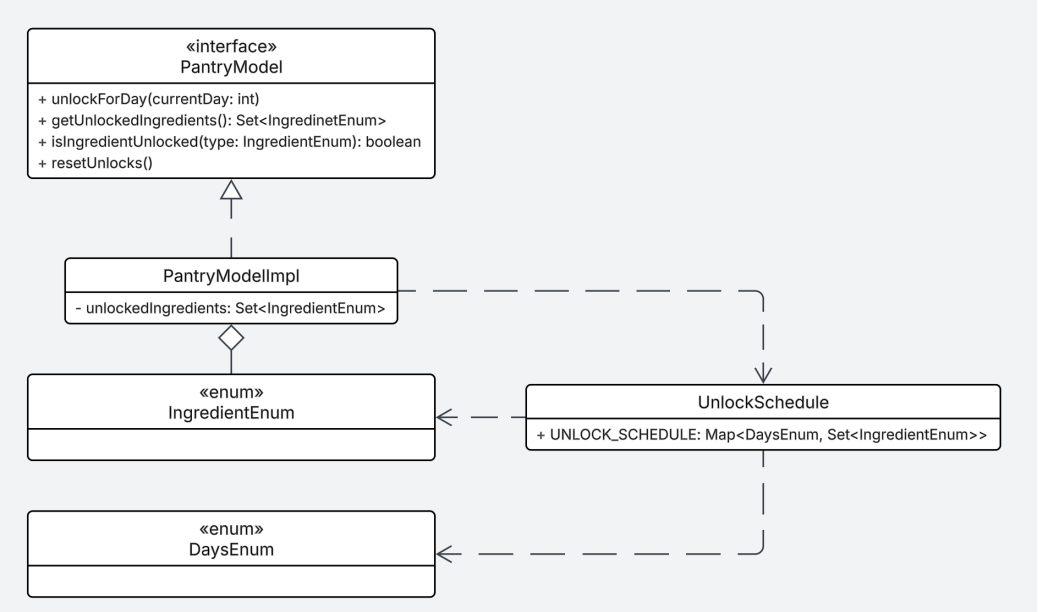
- `Draggable`: indica se uno sprite può essere trascinato direttamente o se ha bisogno di creare una sua copia.
- `Visible`: indica se lo sprite deve essere disegnato o meno.

- `Cloneable`: indica se lo sprite quando viene copiato deve nascondere l'originale e spostarlo al rilascio o se è una vera copia separata dall'originale.
- `Removable`: indica se uno sprite rappresenta un oggetto che arriva dal model e che quindi deve comunicare al rilascio col controller per eventualmente modificare il model.
- `Flipped`: indica lo sprite è stato capovolto in senso verticale o se è dritto.

Creare una classe `SpriteDragManager` che implementa `MouseListener` e `MouseMotionListener` e agisce in pratica da `ActionListener` per gli sprite e che esegue le varie operazioni necessarie nei metodi interessati controllando i flag dello sprite e agendo di conseguenza.

Creare una interfaccia `SpriteDropListener` da implementare nelle view per agire in pratica da `Observer` dei pattern della Go4 e che quindi viene chiamata da `SpriteDragManager` per comunicare alle view eventi particolari e di fatto separando la logica di gioco del drop nelle view da quella pratica di trascinamento.

Sblocco progressivo degli ingredienti



Problema Nel gioco si inizia con la maggior parte degli ingredienti sbloccati e ogni giorno bisogna sbloccare nuovi ingredienti finché non sono tutti

sbloccati. Quindi c'è bisogno di una tabella che specifichi l'ordine di sblocco degli ingredienti e di un model che contenga lo stato degli ingredienti sbloccati.

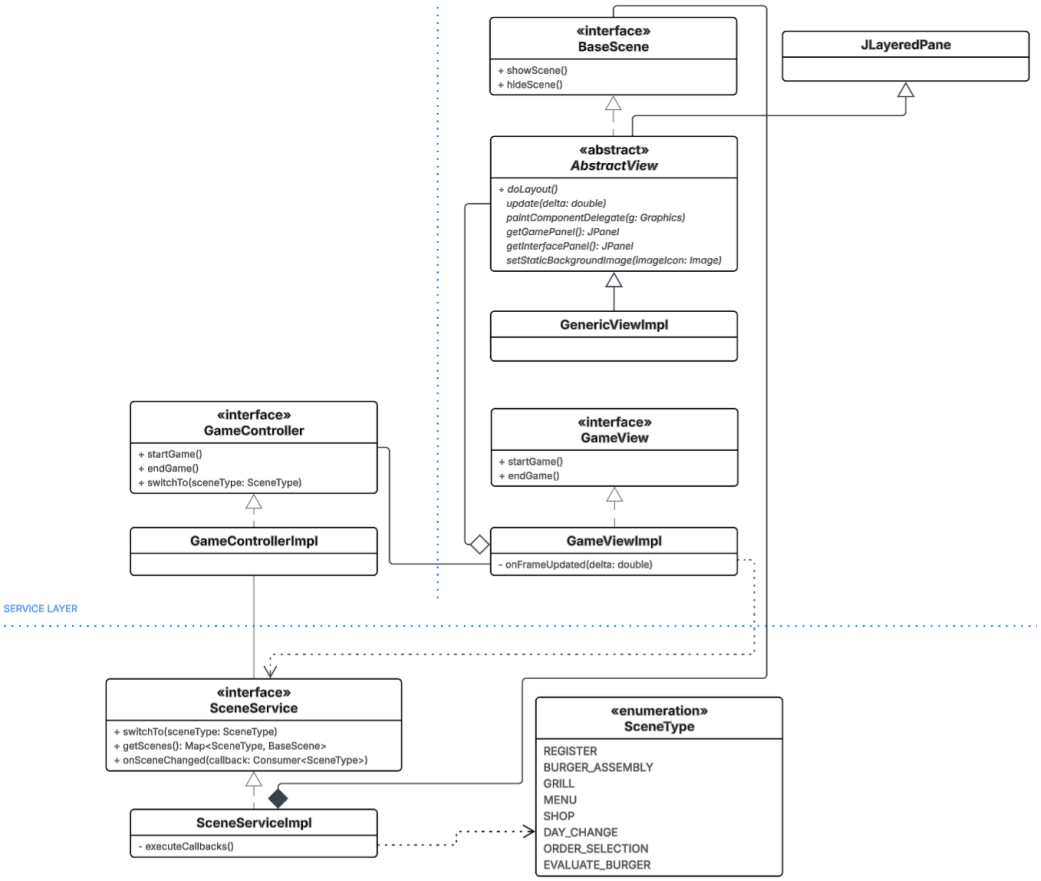
Soluzione Creare una `DaysEnum` e una `IngredientEnum`, la prima per avere delle costanti che rappresentano i giorni ovvero FIRST_DAY, SECOND_DAY, etc., e la seconda per dichiarere i tipi di ingredienti esistenti.

Creare una classe `final UnlockSchedule` con un blocco `static` nel quale dichiara e inizializza una `EnumMap` di `DaysEnum` e di `Set` di `IngredientEnum` nel quale viene rappresentata la tabella di sblocco per ogni giorno degli ingredienti, che quando viene inizializzata viene resa unmodifiable e con un costruttore private per evitare instaurazioni di essa.

Creare una interfaccia `PantryModel` e una sua implementazione `PantryModelImpl` che contiene un `Set` di `IngredientEnum` che contiene gli ingredienti sbloccati e dei metodi per gestire lo sblocco degli ingredienti leggendo la mappa di `UnlockSchedule`.

2.2.3 Kravchuk Paulo

Gestione astratta del concetto di scena per il cambio di componenti View nelle componenti Controller mantenendo separazione dei layer MVC



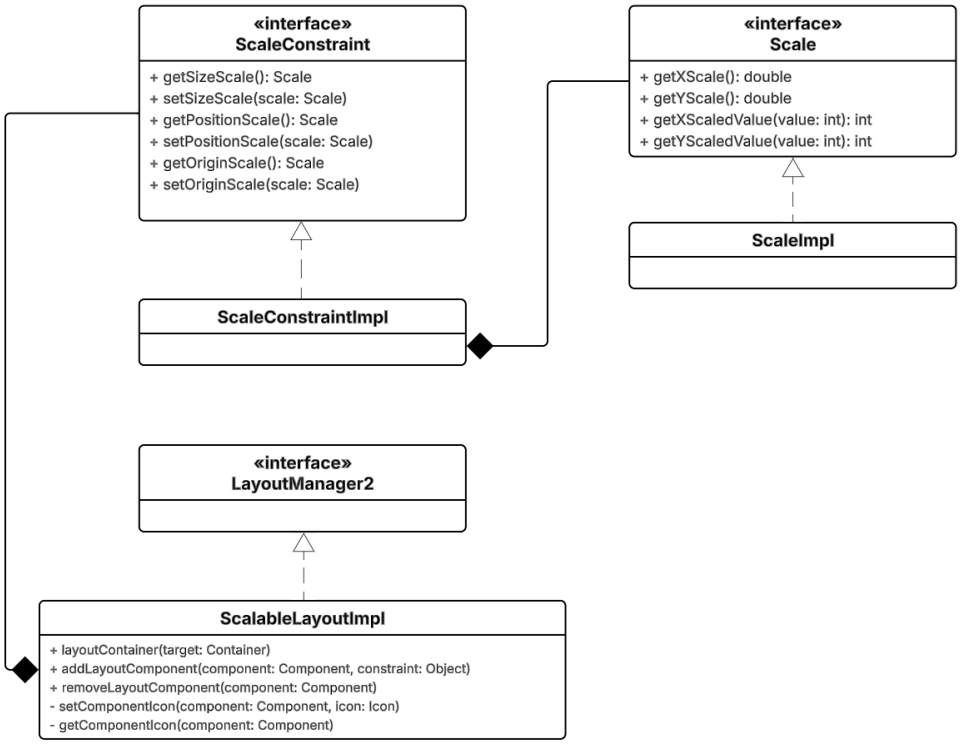
Problema In diversi componenti Controller della nostra applicazione MVC serviva poter indirizzare l'utente verso una scena diversa, in risposta a eventi o input, senza farli dipendere direttamente dalle implementazioni delle componenti View. In caso contrario, si creerebbe dipendenza non necessaria tra layer e accoppiamenti forti che andrebbero a violare i principi dell'architettura MVC.

Soluzione La soluzione prevede l'implementazione di `SceneService`, a funzionalità trasversale tra i layer, che aggiunge un ulteriore livello di astrazione per permettere ai *controller* di indirizzare l'utente verso scene predeterminate senza mai conoscere le classi concrete delle view. Viene associato

ciascun valore dell'enum `SceneType` alla corrispondente implementazione di view. Le view, a loro volta, estendono la classe astratta `AbstractBaseView` (che implementa l'interfaccia `BaseScene` e deriva da `JLayeredPane`). Così i controller si limitano a invocare `switchTo(SceneType)`, mentre `SceneService` si occupa di nascondere la vista corrente (`hideScene()`), mostrare quella nuova (`showScene()`) e notificare eventuali listener. I pattern utilizzati sono:

- `Template Method` per quanto riguarda i metodi astratti `update(double)\verb` e `paintComponentDelegate(Graphics)` di `AbstractBaseView`;
- `Observer` per il meccanismo di callback in `onSceneChanged(Consumer<SceneType>)`

Progettazione di un layout manager a valori percentuali semplice per semplificare il posizionamento dei JComponent

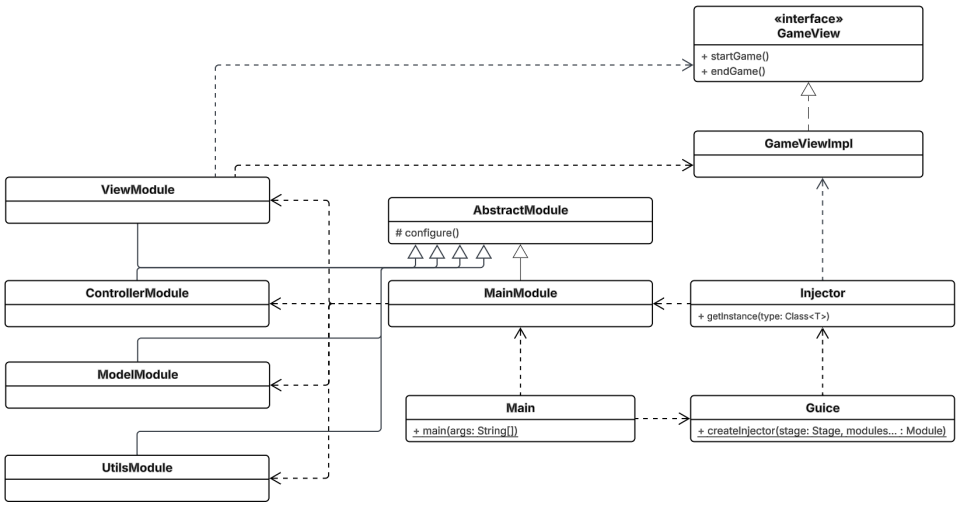


Problema Nelle UI Swing, disporre e ridimensionare i componenti in modo percentuale rispetto al contenitore richiede solitamente di sovrascrivere `doLayout()` o di usare layout manager complessi (`GridBagLayout`, `GroupLayout`), con calcoli pixel-by-pixel e molto codice duplicato. Di conseguenza è difficile

implementare interfacce “responsive” che si adattino al ridimensionamento della finestra mantenendo libertà di posizionamento.

Soluzione È stata creata ScalableLayoutImpl, un’implementazione personalizzata di LayoutManager2 che permette di associare a ogni `Component` uno ScaleConstraint(size, position, origin) espresso in valori [0.0-1.0]. Nell’implementazione il metodo `layoutContainer(...)` calcola automaticamente larghezza, altezza, e posizione rispetto al parent tenendo conto anche del punto di origine. Inoltre, ridimensiona le icone legate a tipi di JComponent che ne supportano la funzionalità. Le alternative a questa soluzione erano poche, Swing è datato e i layout non third-party erano limitanti e non rispettavano le nostre necessità. Inizialmente era stato eseguito un override di `doLayout()` per riposizionare e cambiare misura dei componenti figli in base a valori hard-coded. Questo layout manager come tutti gli altri segue lo Strategy pattern, dove all’aggiunta di un componente ad un parent si può specificare la strategia di layout da usare.

Organizzare le dipendenze senza dover pensare al come strutturale il loro passaggio alle altre classi



Problema Gestire manualmente le dipendenze tra classi in un progetto Java può diventare rapidamente complesso e disordinato. Istanziare oggetti con new sparsi ovunque, trovare il modo per passare a mano riferimenti nei costruttori può rendere il codice difficile da mantenere e testare, specialmente

quando delle classi devono per forza avere un'unica istanza passata a più costruttori per via dello stato contenuto.

Soluzione È stato adottato Google Guice per centralizzare e automatizzare la creazione e l'iniezione delle dipendenze, lasciandogli la risoluzione del grafo di esse. Questo permette anche di definire in modo più facile quale implementazione si vuole dare ad una certa interfaccia, andando a cambiare semplicemente il binding all'interno di un module di Guice. Altre soluzioni come service locator vengono considerate come anti-pattern. Si è rivelato estremamente utile nella gestione di istanze che contenevano stato e quindi per forza non dovevano essere ricreate. Il pattern usato è quello di `Dependency Injection` dove le classi ricevono nel costruttore ciò di cui hanno bisogno e Guice si occupa di collegare il tutto. Il principio architetturale è quello dell'`Inversion of Control` dove il flusso di creazione non è gestito dal nostro codice, ma dal framework di Guice.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il testing automatizzato è stato realizzato, con l'aiuto della dependency JUnit Jupiter, sulla maggior parte dei Controller, tutti i Model e tutti i Service. Sfortunatamente non è stato possibile realizzare i test per le View in Swing per via di costrizioni di tempo.

- `BurgerAssemblyControllerImplTest`: Verifica il corretto funzionamento della classe BurgerAssemblyControllerImpl. In particolare: controlla il corretto funzionamento dell aggiunta e rimozione di ingredienti nel hamburgerOnAssembly contenuto nel model, idem per la lista di cookedPatties, in aggiunta controlla il corretto funzionamento del calculateAccuracy e del changeAccuracy, infine vede se il metodo per prendere gli ingredienti sbloccati e quello per vedere se un tipo di ingrediente è sbloccato si comportano in modo normale.

- `DayChangeControllerImplTest`: Verifica il corretto funzionamento della classe DayChangeControllerImpl. In particolare: controlla che ritorni il giorno corrente corretto e che il metodo per prendere la lista di ingredienti sbloccati nel giorno corrente funzionino.

- `EvaluateBurgerControllerImplTest`: Verifica il corretto funzionamento della classe EvaluateBurgerControllerImpl. In particolare: testa se l'hamburger viene preso correttamente nell'assembly, testa se viene preso correttamente l'odine corrente (sia con valore valido che con null) e se la funzione emptyHamburger crea un hamburger nuovo.

- `GrillControllerImplTest`: Verifica il corretto funzionamento della classe GrillCustomerImpl. In particolare: Prova il corretto funziona-

mento dell'aggiunta e rimozione di un patty dalla griglia e dalla lista di patty pronti, inoltre controlla che il metodo per girare i patty e quello per cuocere i patty sulla griglia si comportino normalmente.

- `MenuControllerImplTest`: Verifica il corretto funzionamento della classe MenuControllerImpl. In particolare: prima di ogni test crea due implementazioni mock di SaveService e GameModel da dare in pasto a MenuControllerImpl, testa che returni correttamente l'index dello save slot corrente e che le DTO SaveState date in input vengano correttamente trasformate in DTO SaveInfo per le componenti View.

- `OrderSelectionControllerImplTest`: Verifica il corretto funzionamento della classe OrderSelectionControllerImpl. In particolare: controlla che gli ordini e l'hamburger vengano presi correttamente, che l'ordine selezionato venga settato e che venga rimosso correttamente l'ultimo ingrediente dell'hamburger.

- `ShopControllerImplTest`: Verifica il corretto funzionamento della classe ShopControllerImpl. In particolare: testa se la funzione isUpgradePurchasable ritorna un valore affidabile, se la buyUpgrade funziona a dovere, che getBalance dia il bilancio corrente e infine verifica che il metodo isUpgradeUnlocked funzioni, i test sono stati fatti creando una mock class di ShopModel.

- `CustomerModelImplTest`: Verifica il corretto funzionamento della classe CustomerModelImpl. In particolare testa se ritorna l'ordine corretto.

- `GameModelTest`: Verifica il corretto funzionamento della classe GameModelImpl. In particolare: prima di tutto controlla che la build funzioni in modo corretto, poi procede testando set e get di balance, hamburgerOnAssembly, cookedPatties, selectedOrder e infine che nextDay aggiorni i dati correttamente.

- `HamburgerModelImplTest`: Verifica il corretto funzionamento della classe HamburgerModelImpl. In particolare: testa se la classe aggiunge correttamente l'ingrediente e rimuove correttamente l'ingrediente in cima. Inoltre controlla che ritorni correttamente una copia di sé.

- `OrderModelImplTest`: Verifica il corretto funzionamento della classe OrderModelImpl. In particolare: testa se ritorna correttamente una copia dell'hamburger, se ritorna il numero di ordine corretto e se ritorna una copia di se.

- `PantryModelImplTest`: Verifica il corretto funzionamento della classe PantryModelImpl. In particolare: parte controllando che la build sia andata bene, per poi procedere testando il metodo che verifica se un tipo di ingrediente è sbloccato o meno, infine controllare il metodo di sblocco degli ingredienti per il giorno corrente e quello di reset degli unlock funzioni.

- `PattyModelImplTest`—: Verifica il corretto funzionamento della classe PattyModelImpl. In particolare: testa set e get del livello di cottura superiore e inferiore, che vengono flippati normalmente e che i metodi copyof, equals e hash siano funzionanti.

- `ShopModelImplTest`: Verifica il corretto funzionamento della classe ShopModelImpl. In particolare: testa se gli upgrade sono inizializzati come bloccati, se blocca e sblocca correttamente gli upgrade, se l'upgrade modifier viene ritornato correttamente e se viene mandata una copia della lista degli upgrade corretta.

- `ResourceImplServiceTest`: Verifica il corretto funzionamento della classe ResourceServiceImpl. In particolare: controlla che il getter per le immagini e quello per i suoni funzionino e restituiscano un'exception per gli argomenti non validi, poi testa se la dispose libera effettivamente le risorse.

- `SaveServiceImplTest`: Verifica il corretto funzionamento della classe SaveServiceImpl. In particolare: testa che il metodo per salvare e per caricare un salvataggio agiscano correttamente prima in modo separato andando a testare anche le exception e poi insieme verificando se input e output combacino, infine controlla che il metodo per ottenere tutti i salvataggi presenti funzioni.

- `SceneServiceImplTest`: Verifica il corretto funzionamento della classe SceneServiceImpl. In particolare: prima di ogni test si creano due implementazioni mock con contatori di BaseScene e una EnumMap¡SceneType, BaseScene¿ da dare in pasto all'implementazione di SceneService, poi controlla il corretto funzionamento dei suoi metodi andando a testare i throw delle exception, se i contatori corrispondono ai valori previsti dopo le chiamate a switchTo, se i callback vengono chiamati correttamente e se l'implementazione espone la reference alla map originale o una copia immutabile.

- `ServiceHelpers`: É una classe statica di utility usata all'interno del test di ResourceServiceImpl e SfxServiceImpl.

- `SfxServiceImplTest`: Verifica il corretto funzionamento della classe SfxServiceImpl. In particolare: controlla che le instanze Clip vengano avviate e fermate correttamente in ambe modalità di partenza (looped e non) e che passando un valore di volume più alto del limite lanci una exception.

3.2 Note di sviluppo

3.2.1 Casamenti Michele

Utilizzo di thread per generazione di clienti automatica

Permalink: `https://github.com/oop-proj-burgeria/OOP24-papas-burgeria/blob/main/src/main/java/it/unibo/papasburgeria/model/impl/CustomerThread.java#L15`

Utilizzo di Google Guice nei costruttori per fare Inject dei parametri

Permalink: `https://github.com/oop-proj-burgeria/OOP24-papas-burgeria/blob/main/src/main/java/it/unibo/papasburgeria/controller/impl/CustomerControlle`
`java#L40`

Utilizzo di Google Guice per indicare una classe Singleton

Permalink: `https://github.com/oop-proj-burgeria/OOP24-papas-burgeria/blob/main/src/main/java/it/unibo/papasburgeria/model/impl/RegisterModelImpl.java#18`

3.2.2 Ranzari Gabriele

Utilizzo di Google Guice nei costruttori per fare Inject dei parametri

Permalink: `https://github.com/oop-proj-burgeria/OOP24-papas-burgeria/blob/main/src/main/java/it/unibo/papasburgeria/controller/impl/BurgerAssemblyCon`
`java#L42`

Utilizzo di Google Guice per indicare una classe Singleton

Permalink: https://github.com/oop-proj-burgeria/OOP24-papas-burgeria/blob/main/src/main/java/it/unibo/papasburgeria/model/impl/PantryModelImpl.java#L23

Utilizzo di lambda expression molto semplice per ogni bottone creato come nel seguente esempio

Permalink: https://github.com/oop-proj-burgeria/OOP24-papas-burgeria/blob/main/src/main/java/it/unibo/papasburgeria/view/impl/OrderSelectionViewImpl.java#L88

Metodo per fare il flip verticale delle immagini riadattato visto sul web

Permalink: https://github.com/oop-proj-burgeria/OOP24-papas-burgeria/blob/main/src/main/java/it/unibo/papasburgeria/view/impl/components/SpriteImpl.java#264

3.2.3 Kravchuk Paulo

Uso di jackson per mappare i DTO in JSON non protetti nella gestione dei salvataggi

Permalink: https://github.com/oop-proj-burgeria/OOP24-papas-burgeria/blob/17d0eb117ae0d732d72b44bf4a14905ba6febede/src/main/java/it/unibo/papasburgeria/utils/impl/saving/SaveServiceImpl.java#L70

Uso di Google Guice per realizzare l'astrazione delle scene creando una map di binder

Permalink: https://github.com/oop-proj-burgeria/OOP24-papas-burgeria/blob/17d0eb117ae0d732d72b44bf4a14905ba6febede/src/main/java/it/unibo/papasburgeria/di/UtilsModule.java#L35

Uso di lambda come callback

Permalink: https://github.com/oop-proj-burgeria/OOP24-papas-burgeria/blob/17d0eb117ae0d732d72b44bf4a14905ba6febede/src/main/java/it/unibo/papasburgeria/view/impl/GameViewImpl.java#L199

Uso di Clip per la gestione degli sfx

Permalink: https://github.com/oop-proj-burgeria/OOP24-papas-burgeria/blob/17d0eb117ae0d732d72b44bf4a14905ba6febede/src/main/java/it/unibo/papasburgeria/utils/impl/resource/ResourceServiceImpl.java#L62

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Casamenti Michele

Il lavoro di gruppo è stata per me una nuova esperienza. Non mi ero mai ritrovato a condividere un progetto con altre persone. L'esperienza è stata piacevole e il gruppo ha lavorato in modo coordinato. Peccato per il membro del gruppo che a un certo punto se ne è andato, e mi sono ritrovato a fare parte del suo lavoro. Fortunatamente, la mole di lavoro aggiuntiva non è stata troppa. Per il futuro devo decisamente suddividere il lavoro in più giornate, in quanto mi sono ritrovato gli ultimi giorni a scrivere molto codice. Una volta compreso come muovermi nel codice, e utilizzando il paradigma MCV il lavoro è stato molto più facile. Non avevo mai realizzato un gioco a questo livello. Da questa esperienza ho imparato molto, dall'importanza della divisione dei compiti al management del tempo. La mia porzione di codice può decisamente essere resa più ordinata. Ci sono richiami a classi e funzioni che penso sarebbero potuti essere organizzati meglio, e penso che le interfacce e le modellazioni potrebbero essere più espandibili. Mi è piaciuto creare delle classi view e crearmi dei modelli, ma ci ho messo un po' a capire come usare le injections e i controller. Se ripartissi da zero, sarei in grado di rifarlo da solo. Una funzionalità opzionale che non ho implementato è stata quella della pazienza, e se non fosse per il tempo sarei riuscito a implementarlo. L'utilizzo di google guice mi ha aperto un mondo sul come fare injection in classi singleton, e mi ha aperto gli occhi su un modo nuovo di programmare.

In conclusione, l'esperienza è stata piacevole ma anche stressante. Vorrei aver distribuito meglio la mole di lavoro e aver implementato meglio nel codice gli insegnamenti fatti a lezione. Vorrei aver usato di più le stream, le lambda expression e gli Optional invece di null.

4.1.2 Ranzari Gabriele

Ho visto il progetto come bella esperienza, non è stata la prima volta che sviluppo un progetto in gruppo, né tantomeno in Java, però quello che avevo fatto alle superiori era molto più buttato come codice, quindi ho comunque riscontrato diverse difficoltà, intanto ad usare il modello MVC in quanto richiede un livello di ragionamento aggiuntivo che non avevo mai dovuto fare. Inoltre è stata la mia primissima volta ad usare Google Guice. Per quanto riguarda il codice che ho scritto mi sento discretamente soddisfatto in quanto sento che se adesso riiniziassi da capo il progetto riuscirei a strutturarlo in un modo molto migliore, in quanto mi sono ritrovato a modificare il codice svariate volte per riadattarlo e trovo che classi come Sprite e SpriteDragManager le avrei potute gestire in modo molto migliore, e questo penso sia dovuto al fatto che fatico molto ad analizzare quello che mi servirà in futuro e al pensare di dover scrivere una classe/metodo in modo generale e non in modo specifico a come mi serve adesso. La cosa che vorrei rifare di più è togliere l'ingredient dalla classe sprite in quanto inizialmente l'avevo pensata solo per gli ingredienti ma alla fine mi sono ritrovato ad usarla per anche per gestire il disegno e drag del foglio dell'ordine che per farlo funzionare ho dovuto associargli un ingrediente che trovo super sbagliato.

Per quanto riguarda la gestione del tempo penso di essermela cavata bene finchè il nostro compagno non è uscito dal gruppo, in quanto mi sono ritrovato a dover fare classi che non avevo neanche pensato come fare e con gli esami di mezzo non ho avuto il tempo per sistemare il mio codice come avrei potuto.

In conclusione l'esperienza è stata piacevole e molto istruttiva, però mi è sembrato più un dovere che un vero piacere programmare il gioco rispetto a quello che feci alle superiori, trovo che questo sentimento sia dovuto al fatto che (com'è giusto che sia) importava in pratica solo il codice e non l'aspetto grafico o di quanto fosse curato il gameplay del progetto il che mi spiace.

4.1.3 Kravchuk Paulo

Non è il primo progetto che affronto, ho lavorato e sto continuamente lavorando con diversi team, ma come sempre è stata un'esperienza interessante, specialmente la parte del potere sia apprendere cose nuove dagli altri che condividere quello che si sa con gli altri membri del gruppo. Ho già avuto a che fare con l'architettura MVC per quanto riguarda le applicazioni web in ASP .NET ed è stato bello poter applicare la stessa in una diversa sfaccettatura (sia per il tipo di progetto, che è un gioco, ma anche per il linguaggio diverso).

Nonostante le difficoltà iniziali nel definire l'architettura del progetto, una volta gettate le fondamenta è stato molto più semplice sviluppare nuove funzionalità. Mi sono concentrato principalmente sul semplificare il quanto possibile lo sviluppo, sia per me che per gli altri membri del team, così da ridurre le difficoltà tecniche e al contrario favorire la realizzazione delle varie componenti.

C'è stato un problema con un membro, che poi ha abbandonato il team per via di problemi personali, causando uno spostamento di una mole significativa di lavoro sui membri restanti, quindi guardando indietro probabilmente cercherei di fare in modo che l'organizzazione del team sia più solida contro situazioni del genere.

In conclusione è stata un'esperienza bella, stressante ma formativa.

Appendice A

Guida utente

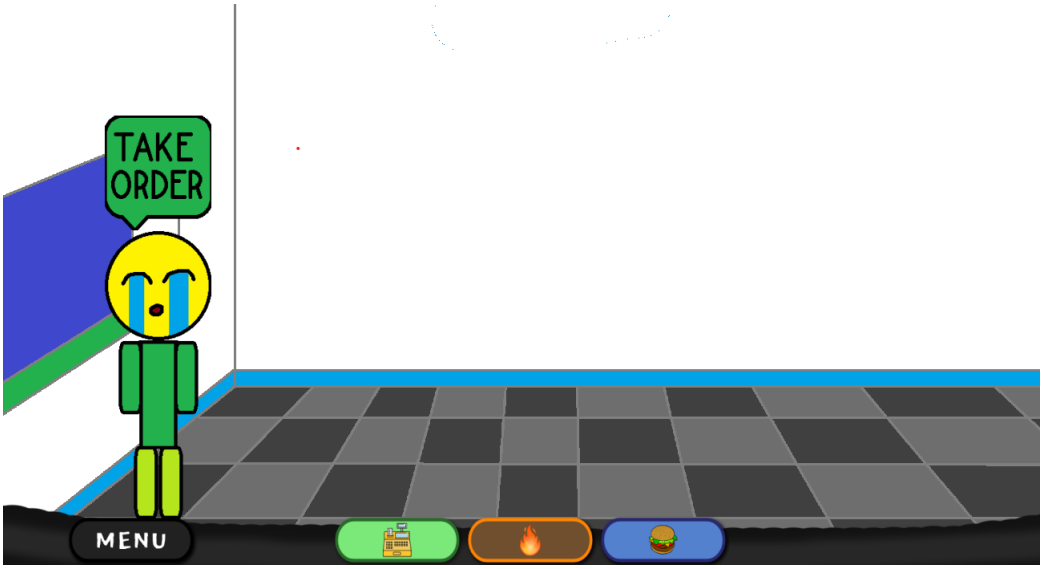
A.1 Menu

All'avvio del programma verrà aperta la schermata principale. Si prema il tasto "play". Poi selezionare lo slot di salvataggio sul quale si vuole proseguire/iniziare la partita.

A.2 Register

Appena iniziata la partita, il giocatore verrà mandato sulla schermata della cassa. Si vedrà un personaggio con un iconcina a forma di fumetto verde con su scritto "Take Order". Si prema l'iconcina verde per prendere l'ordine del personaggio. Ripetere finché i clienti non sono finiti. I clienti arrivano

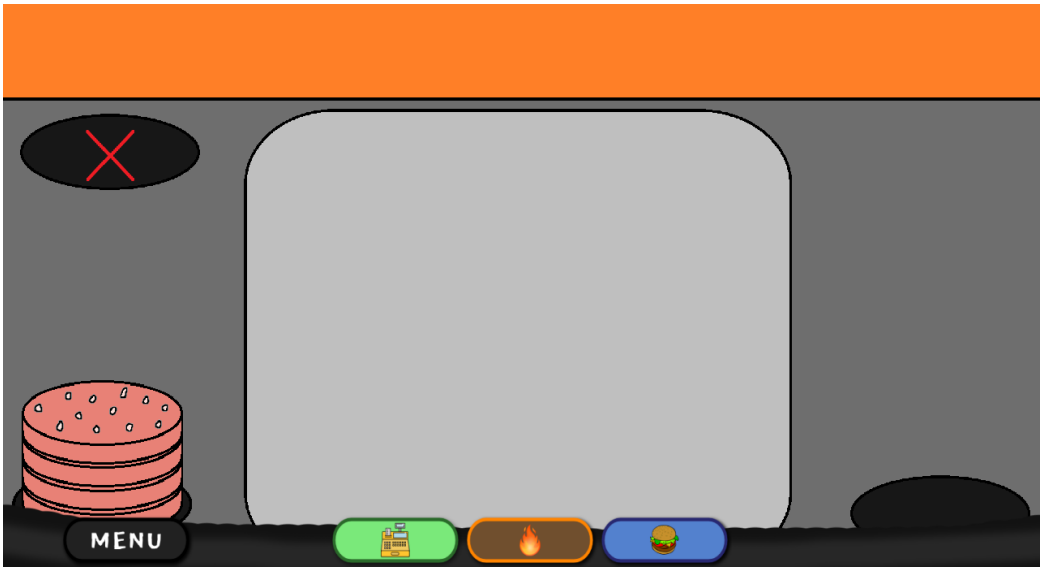
periodicamente da soli, se entrambe le file sono vuote, presto arriverà un altro cliente.



Per cambiare da una schermata all'altra si premano i bottoni nella barra in basso. I bottoni indicati sono (da sinistra a destra): visualizza schermata menù, visualizza schermata della cassa (verde con una cassa disegnata), visualizza schermata della griglia (rosso con una fiamma) e visualizza schermata dell'assemblaggio degli hamburger (blu con un hamburger).

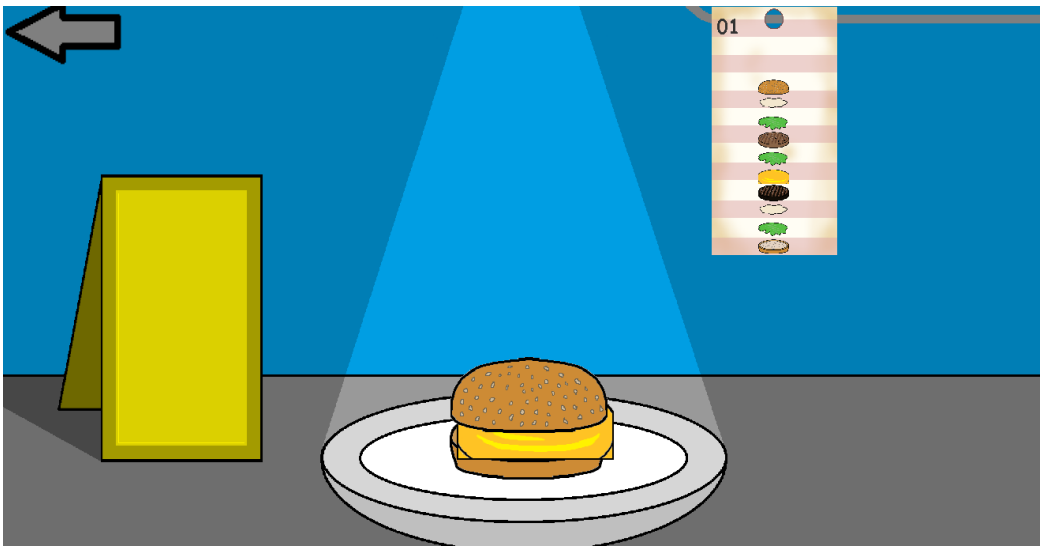
A.3 Grill

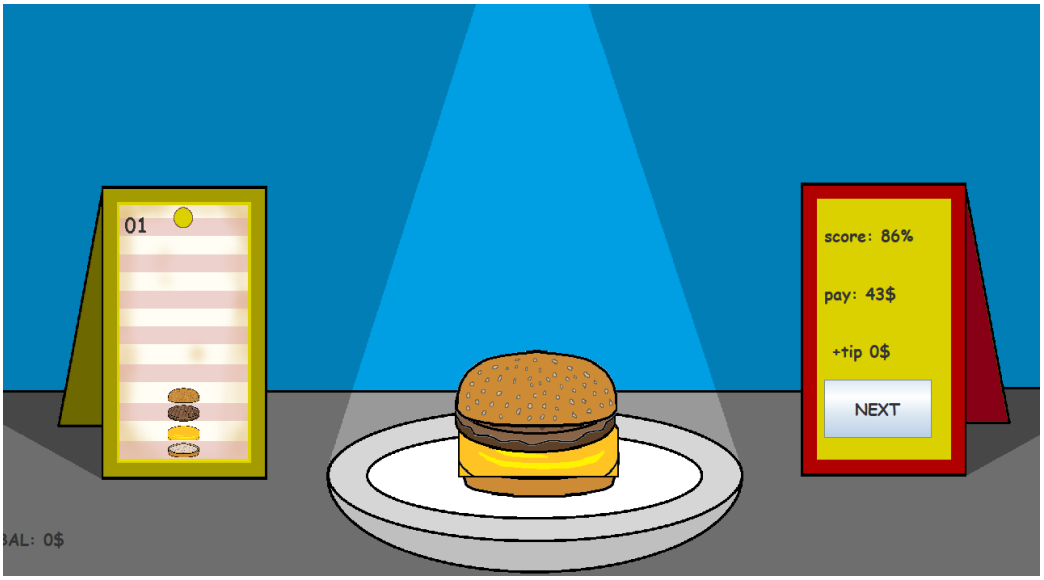
Nella schermata della griglia si cucinano i patty che poi verranno portati nella build station. Per cucinare il patty trascinarlo sulla griglia e aspettare che si cuocia. Vanno cotti entrambi i lati. Per girare il patty cliccarci sopra. Quando il patty è cotto come desiderato trascinarlo in basso a destra sul cerchio nero. Se il patty è stato cotto troppo trascinarlo in alto a sinistra, nell'icona con la 'X' rossa. Quando verrà rilasciato scomparirà. Quando si passerà alla build station si potrà accedere ai patty cucinati.



A.4 Assembly and Evaluation

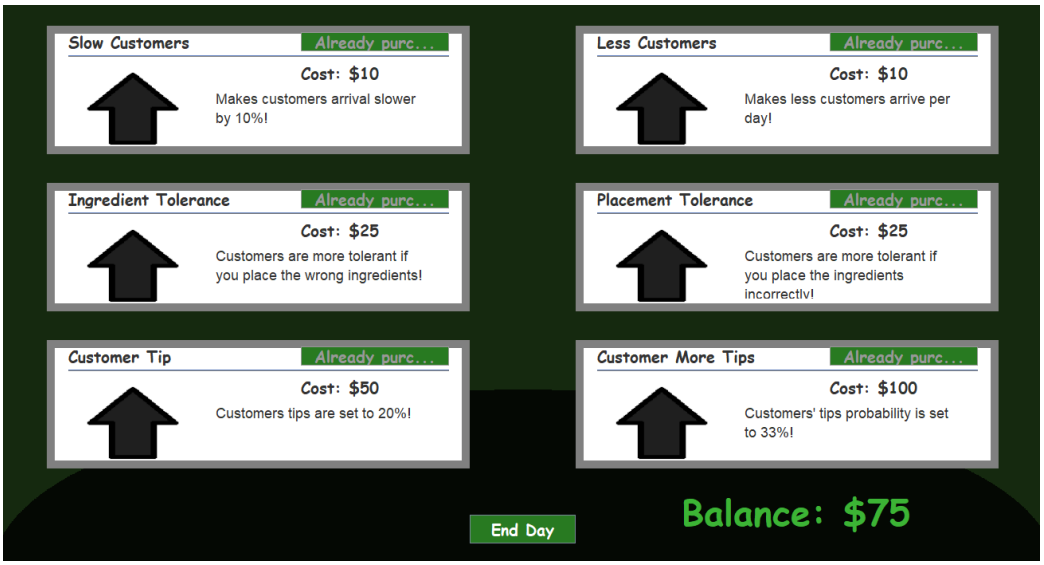
La schermata costruzione hamburger si presenta con un piatto al centro, gli ingredienti in alto a sinistra e gli ordini in alto a destra. Per costruire un hamburger trascinare gli ingredienti sul piatto. Quando verrà posizionato l'ultimo ingrediente (il panino coi semi di sesamo) trascinare sul piedistallo giallo l'ordine che si desidera consegnare. Infine premere il tasto "next" sul piedistallo rosso.


01


01
score: 86%
pay: 43$
+tip 0$
NEXT
AL: 0$

A.5 Shop

Quando sono stati serviti tutti i clienti, la giornata sarà finita e verrà mostrato lo shop dove si può selezionare uno o più upgrade da comprare. Appena finiti gli acquisti, premere il bottone verde "end day" per terminare la giornata.



A.6 ChangeDay

Finita la giornata, passerà ad una scena dove mostra quali ingredienti sono stati sbloccati. Premere il tasto "New Day" per cominciare la nuova giornata.